Chapter TEN
Java Built-In Lambda Interfaces

Exam Objectives

Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier.
Develop code that uses primitive versions of functional interfaces.
Develop code that uses binary versions of functional interfaces.
Develop code that uses the UnaryOperator interface.

Why built-in interfaces?

A lambda expression must correspond to one functional interface.

We can use any interface as a lambda expression as long as the interface only contains one abstract method.

We also saw that the Java 8 API has many functional interfaces that we can use to construct lambda expressions, like `java.lang.Runnable` or `java.lang.Comparable`.

However, Java 8 contains new functional interfaces to work specifically with lambda expressions, covering the most common scenarios usages.

For example, two common scenarios are to filter things based on a particular condition and test for some condition on the properties of an object.

In the previous chapters, we used:

```
interface Searchable {
    boolean test(Car c);
}
```

But the problem is that we have to write an interface like that in each program that uses it (or link a library that contains it).

Luckily, an interface that does the same but accepts any object type already exists in the language.

The new functional interfaces are located inside the `java.util.function` package.

There are five of them:

- **`Predicate<T>`**
- **`Consumer<T>`**
- **`Function<T, R>`**
- **`Supplier<T>`**
- **`UnaryOperator<T>`**

Where `T` and `R` represent generic types (`T` represents a parameter type and `R` the return type).

Also, they also have specializations for the cases where the input parameter is a primitive type (actually just for `int` , `long` , `double` , and `boolean` just in the case of `Supplier`), for example:

- **`IntPredicate`**
- **`LongConsumer`**
- **`BooleanSupplier`**

Where the name is preceded by the appropriate primitive type.

Plus, four of them have binary versions, which means they take two parameters instead of one:

- **`BiPredicate<L, R>`**
- **`BiConsumer<T, U>`**
- **`BiFunction<T, U, R>`**
- **`BinaryOperator<T>`**

Where `T` , `U` , and `R` represent generic types (`T` and `U` represent parameter types and `R` the return type).

The following tables show the complete list of interfaces.

You don't have to memorize them, just try to understand them.

In the following pages, each interface will be explained.

Functional Interface	Primitive Versions

`Predicate<T>`	`IntPredicate`
	`LongPredicate`
	`DoublePredicate`

`Consumer<T>`	`IntConsumer`
	`LongConsumer`
	`DoubleConsumer`

`Function<T, R>`	`IntFunction<R>`
	`IntToDoubleFunction`
	`IntToLongFunction`
	`LongFunction<R>`
	`LongToDoubleFunction`
	`LongToIntFunction`
	`DoubleFunction<R>`
	`DoubleToIntFunction`
	`DoubleToLongFunction`
	`ToIntFunction<T>`
	`ToDoubleFunction<T>`
	`ToLongFunction<T>`

`Supplier<T>`	`BooleanSupplier`
	`IntSupplier`
	`LongSupplier`
	`DoubleSupplier`

`UnaryOperator<T>`	`IntUnaryOperator`
	`LongUnaryOperator`
	`DoubleUnaryOperator`

Functional Interface	Primitive Versions
`BiPredicate<L, R>`	

`BiConsumer<T, U>`	`ObjIntConsumer<T>`
	`ObjLongConsumer<T>`
	`ObjDoubleConsumer<T>`

`BiFunction<T, U, R>`	`ToIntBiFunction<T, U>`
	`ToLongBiFunction<T, U>`
	`ToDoubleBiFunction<T, U>`

`BinaryOperator<T>`	`IntBinaryOperator`
	`LongBinaryOperator`
	`DoubleBinaryOperator`

Predicate

A predicate is a statement that may be true or false depending on the values of its variables.

This functional interface can be used anywhere you need to evaluate a `boolean` condition.

This how the interface is defined:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // Other default and static methods
    // ...
}
```

So the functional descriptor (method signature) is:

```
T -> boolean
```

Here's an example using an anonymous class:

```
Predicate<String> startsWithA = new Predicate<String>() {
    @Override
    public boolean test(String t) {
        return t.startsWith("A");
    }
};
boolean result = startsWithA.test("Arthur");
```

And with a lambda expression:

```
Predicate<String> startsWithA = t -> t.startsWith("A");
boolean result = startsWithA.test("Arthur");
```

This interface also has the following default methods:

```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
default Predicate<T> negate()
```

These methods return a composed `Predicate` that represents a short-circuiting logical **AND** and **OR** of this predicate and another and its logical negation.

Short-circuiting means that the other predicate won't be evaluated if the value of the first predicate can predict the result of the operation (if the first predicate return false in the case of **AND** or if it returns true in the case of **OR**).

These methods are useful to combine predicates and make the code more readable, for example:

```
Predicate<String> startsWithA = t -> t.startsWith("A");
Predicate<String> endsWithA = t -> t.endsWith("A");
boolean result = startsWithA.and(endsWithA).test("Hi");
```

Also there's a `static` method:

```
static <T> Predicate<T> isEqual(Object targetRef)
```

That returns a `Predicate` that tests if two arguments are equal according to `Objects.equals(Object, Object)`.

There are also primitive versions for `int`, `long` and `double`. They don't extend from `Predicate`.

For example, here's the definition of `IntPredicate`:

```
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);
    // And the default methods: and, or, negate
}
```

So instead of using:

```
Predicate<Integer> even = t -> t % 0 == 1;
boolean result = even.test(5);
```

You can use:

```
IntPredicate even = t -> t % 0 == 1;
boolean result = even.test(5);
```

Why?

Just to avoid the conversion from `Integer` to `int` and work directly with primitive types.

Notice that these primitive versions don't have a generic type. Due to the way generics are implemented, parameters of the functional interfaces can be bound only to object types.

Since the conversion from the wrapper type (`Integer`) to the primitive type (`int`) uses more memory and comes with a performance cost, Java provides these versions to avoid autoboxing operations when inputs or outputs are primitives.

Consumer

A consumer is an operation that accepts a single input argument and returns no result; it just execute some operations on the argument.

This how the interface is defined:

```
@FunctionalInterface
public interface Consumer<T> {
     void accept(T t);
     // And a default method
     // ...
}
```

So the functional descriptor (method signature) is:

```
T -> void
```

Here's an example using an anonymous class:

```
Consumer<String> consumeStr = new Consumer<String>() {
     @Override
     public void accept(String t) {
         System.out.println(t);
     }
};
consumeStr.accept("Hi");
```

And with a lambda expression:

```
Consumer<String> consumeStr = t -> System.out.println(t);
consumeStr.accept("Hi");
```

This interface also has the following default method:

```
default Consumer<T> andThen(Consumer<? super T> after)
```

This method returns a composed `Consumer` that performs, in sequence, the operation of the consumer followed by the operation of the parameter.

These methods are useful to combine `Consumer` s and make the code more readable, for example:

```
Consumer<String> first = t ->
         System.out.println("First:" + t);
Consumer<String> second = t ->
         System.out.println("Second:" + t);
first.andThen(second).accept("Hi");
```

The output is:

```
First: Hi
Second: Hi
```

Look how both `Consumer` s take the same argument and the order of execution.

There are also primitive versions for `int` , `long` and `double` . They don't extend from `Consumer` .

For example, here's the definition of `IntConsumer` :

```
@FunctionalInterface
public interface IntConsumer {
     void accept(int value);
     default IntConsumer andThen(IntConsumer after) {
         // ...
     }
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, t -> System.out.println(t));
//...
void printList(int[] a, Consumer<Integer> c) {
     for(int i : a) {
         c.accept(i);
     }
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, t -> System.out.println(t));
//...
void printList(int[] a, IntConsumer c) {
     for(int i : a) {
         c.accept(i);
     }
}
```

Function

A function represents an operation that takes an input argument of a certain type and produces a result of another type.

A common use is to convert or transform from one object to another.

This how the interface is defined:

```
@FunctionalInterface
public interface Function<T, R> {
     R apply(T t);
     // Other default and static methods
     // ...
}
```

So the functional descriptor (method signature) is:

```
T -> R
```

Assuming a method:

```
void round(double d, Function<Double, Long> f) {
     long result = f.apply(d);
     System.out.println(result);
}
```

Here's an example using an anonymous class:

```
round(5.4, new Function<Double, Long>() {
     Long apply(Double d) {
         return Math.round(d);
     }
});
```

And with a lambda expression:

```
round(5.4, d -> Math.round(d));
```

This interface also has the following default methods:

```
default <V> Function<V,R> compose(
         Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(
         Function<? super R,? extends V> after)
```

The difference between these methods is that `compose` applies the function represented by the parameter first, and its result serves as the input to the other function. `andThen` first applies the function that calls the method, and its result acts as the input of the function represented by the parameter.

For example:

```
Function<String, String> f1 = s -> {
     return s.toUpperCase();
};
Function<String, String> f2 = s -> {
     return s.toLowerCase();
};
 System.out.println(f1.compose(f2).apply("Compose"));
System.out.println(f1.andThen(f2).apply("AndThen"));
```

The output is:

```
COMPOSE
andthen
```

In the first case, `f1` is the last function to be applied.
In the second case, `f2` is the last function to be applied.

Also there's a `static` method:

```
static <T> Function<T, T> identity()
```

That returns a function that always returns its input argument.

In the case of primitive versions, they also apply to `int` , `long` and `double` , but there are more combinations than the previous interfaces:

- To indicate that the function returns a generic type and takes a primitive argument, the interface is named **XXXFunction**, for example, `IntFunction` :

```
@FunctionalInterface
public interface IntFunction<R> {
     R apply(int value);
}
```

- To indicate that the function returns a primitive type and takes a generic argument, the interface is named **ToXXXFunction**, for example, `ToIntFunction` :

```
@FunctionalInterface
public interface ToIntFunction<T> {
     int applyAsInt(T value);
}
```

- To indicate that the function takes a primitive argument and returns another primitive type, the interface is named **XXXToYYYFunction**, where **XXX** is the argument type and **YYY** is the return type, for example, `IntToDoubleFunction` :

```
@FunctionalInterface
public interface IntToDoubleFunction {
     double applyAsDouble(int value);
}
```

Remember that these interfaces are for convenience, to work directly with primitives, for example:

`DoubleFunction<R>` instead of `Function<Double, R>`
`ToLongFunction<T>` instead of `Function<T, Long>`
`IntToLongFunction` instead of `Function<Integer, Long>`

Supplier

A supplier does the opposite of a consumer, it takes no arguments and only returns some value.

This how the interface is defined:

```
@FunctionalInterface
public interface Supplier<T> {
     T get();
}
```

So the functional descriptor (method signature) is:

```
() -> T
```

Here's an example using an anonymous class:

```
String t = "One";
Supplier<String> supplierStr = new Supplier<String>() {
     @Override
     public String get() {
         return t.toUpperCase();
     }
};
System.out.println(supplierStr.get());
```

And with a lambda expression:

```
String t = "One";
Supplier<String> supplierStr = () -> t.toUpperCase();
System.out.println(supplierStr.get());
```

This interface doesn't define default methods.

There are also primitive versions for `int` , `long` and `double` and `boolean` . They don't extend from `Supplier` .

For example, here's the definition of `BooleanSupplier` :

```
@FunctionalInterface
public interface BooleanSupplier {
     boolean getAsBoolean();
}
```

That can be used instead of `Supplier` .

UnaryOperator

`UnaryOperator` is just a specialization of the `Function` interface (in fact, this interface extends from it) for when the argument and the result are of the same type.

This how the interface is defined:

```
@FunctionalInterface
public interface UnaryOperator<T>
         extends Function<T, T> {
     // Just the identity
     // method is defined
}
```

So the functional descriptor (method signature) is:

```
T -> T
```

Here's an example using an anonymous class:

```
UnaryOperator<String> uOp = new UnaryOperator<String>() {
     @Override
public String apply(String t) {
         return t.substring(0,2);
     }
};
System.out.println(uOp.apply("Hello"));
```

And with a lambda expression:

```
UnaryOperator<String> uOp = t -> t.substring(0,2);
System.out.println(uOp.apply("Hello"));
```

This interface inherits the default methods of the `Function` interface:

```
default <V> Function<V,R> compose(
         Function<? super V,? extends T> before)
```

```
default <V> Function<T,V> andThen(
         Function<? super R,? extends V> after)
```

And just defines the `static` method `identity()` for this interface (since `static` methods are not inherited):

```
static <T> UnaryOperator<T> identity()
```

That returns an `UnaryOperator` that always returns its input argument.

There are also primitive versions for `int`, `long` and `double`. They don't extend from `UnaryOperator`.

For example, here's the definition of `IntUnaryOperator`:

```
@FunctionalInterface
public interface IntUnaryOperator {
     int applyAsInt(int value);
         // Definitions for compose, andThen, and identity
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);
//...
int sumNumbers(int[] a, UnaryOperator<Integer> unary) {
         int sum = 0;
         for(int i : a) {
                 sum += unary.apply(i);
         }
         return sum;
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);
//...
int sumNumbers(int[] a, IntUnaryOperator unary) {
         int sum = 0;
         for(int i : a) {
                 sum += unary.applyAsInt(i);
         }
         return sum;
}
```

BiPredicate

This interface represents a predicate that takes two arguments.

This how the interface is defined:

```
@FunctionalInterface public interface BiPredicate<T, U> {
     boolean test(T t, U u);
     // Default methods are defined also
}
```

So the functional descriptor (method signature) is:

```
(T, U) -> boolean
```

Here's an example using an anonymous class:

```
BiPredicate<Integer, Integer> divisible =
         new BiPredicate<Integer, Integer>() {
     @Override
     public boolean test(Integer t, Integer u) {
         return t % u == 0;
     }
};
boolean result = divisible.test(10, 5);
```

And with a lambda expression:

```
BiPredicate<Integer, Integer> divisible =
        (t, u) -> t % u == 0;
boolean result = divisible.test(10, 5);
```

This interface defines the same default methods of the Predicate interface, but with two arguments:

```
default BiPredicate<T, U> and(
         BiPredicate<? super T, ? super U> other)
default BiPredicate<T, U> or(
         BiPredicate<? super T, ? super U> other)
default BiPredicate<T, U> negate()
```

This interface doesn't have primitive versions.

BiConsumer

This interface represents a consumer that takes two arguments (and don't return a result).

This how the interface is defined:

```
@FunctionalInterface
public interface BiConsumer<T, U> {
     void accept(T t, U u);
     // andThen default method is defined
}
```

So the functional descriptor (method signature) is:

```
(T, U) -> void
```

Here's an example using an anonymous class:

```
BiConsumer<String, String> consumeStr =
         new Consumer<String, String>() {
     @Override
     public void accept(String t, String u) {
         System.out.println(t + " " + u);
     }
 };
consumeStr.accept("Hi", "there");
```

And with a lambda expression:

```
BiConsumer<String> consumeStr =
            (t, u) -> System.out.println(t + " " + u);
consumeStr.accept("Hi", "there");
```

This interface also has the following default method:

```
default BiConsumer<T, U> andThen(
             BiConsumer<? super T, ? super U> after)
```

This method returns a composed `BiConsumer` that performs, in sequence, the operation of the consumer followed by the operation of the parameter.

As in the case of a `Consumer`, these methods are useful to combine `BiConsumer` s and make the code more readable, for example:

```
BiConsumer<String, String> first = (t, u) -> System.out.println(t.toUpperCase() + u.tol
BiConsumer<String, String> second = (t, u) -> System.out.println(t.toLowerCase() + u.tc
first.andThen(second).accept("Again", " and again");
```

The output is:

```
AGAIN AND AGAIN again and again
```

There are also primitive versions for `int`, `long` and `double`. They don't extend from `BiConsumer`, and instead of taking two `int` s, for example, they take one object and a primitive value as a second argument. So the naming convention changes to **ObjXXXConsumer**, where **XXX** is the primitive type. For example, here's the definition of `ObjIntConsumer`:

```
@FunctionalInterface
public interface ObjIntConsumer<T> {
     void accept(T t, int value);
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));
//...
void printList(int[] a, BiConsumer<String, Integer> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));
//...
void printList(int[] a, ObjIntConsumer<String> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

BiFunction

This interface represents a function that takes two arguments of different type and produces a result of another type.

This how the interface is defined:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
     R apply(T t, U u); // Other default and static methods
     // ...
}
```

So the functional descriptor (method signature) is:

```
(T, U) -> R
```

Assuming a method:

```
void round(
        double d1, double d2, BiFunction<Double, Double, Long> f) {
         long result = f.apply(d1, d2);
         System.out.println(result);
}
```

Here's an example using an anonymous class:

```
round(5.4, 3.8, new BiFunction<Double, Double, Long>() {
         Long apply(Double d1, Double d2) {
                 return Math.round(d1 + d2);
         }
});
```

And with a lambda expression:

```
round(5.4, 3.8, (d1, d2) -> Math.round(d1, d2));
```

This interface, unlike `Function`, has only one default method:

```
default <V> Function<T, V> andThen(
                 Function<? super R, ? extends V> after)
```

That returns a composed function that first applies the function that calls `andThen` to its input, to finally apply the function represented by the argument to the result.

This interface also has less primitive versions than `Function`. It only has the versions that take generic types as arguments and return `int`, `long` and `double` primitive types, with the naming convention **ToXXXBiFunction**, where XXX is the primitive type.

For example, here's the definition of `ToIntBiFunction`:

```
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
     int applyAsInt(T t, U u);
}
```

That replaces `BiFunction`.

BinaryOperator

This interface is a specialization of the `BiFunction` interface (in fact, this interface extends from it) for when the arguments and the result are of the same type.

This how the interface is defined:

```
@FunctionalInterface
public interface BinaryOperator<T>
             extends BiFunction<T,T,T> {
     // Two static method are defined
}
```

So the functional descriptor (method signature) is:

```
(T, T) -> T
```

Here's an example using an anonymous class:

```
BinaryOperator<String> binOp =  new BinaryOperator<String>() {
     @Override
     public String apply(String t, String u) {
         return t.concat(u);
     }
};
System.out.println(binOp.apply("Hello", " there"));
```

And with a lambda expression:

```
BinaryOperator<String> binOp = (t, u) -> t.concat(u);
System.out.println(binOp.apply("Hello", " there"));
```

This interface inherits the default method of the `BiFunction` interface:

```
default <V> Function<T, V> andThen(
                 Function<? super R, ? extends V> after)
```

And defines two new `static` methods:

```
static <T> BinaryOperator<T> minBy(
                  Comparator<? super T> comparator)
static <T> BinaryOperator<T> maxBy(
                  Comparator<? super T> comparator)
```

That return a `BinaryOperator` , which returns the lesser or greater of two elements according to the specified `Comparator` .

Here's a simple example:

```
BinaryOperator<Integer> biOp =
             BinaryOperator.maxBy(Comparator.naturalOrder());
System.out.println(biOp.apply(28, 8));
```

As you can see, these methods are just a wrapper to execute a `Comparator` .

`Comparator.naturalOrder()` returns a `Comparator` that compares `Comparable` objects in natural order. To execute it, we just call the `apply()` method with the two arguments needed by the `BinaryOperator`. Unsurprisingly, the output is:

```
28
```

There are also primitive versions for `int`, `long` and `double`, where the two arguments and the return type are of the same primitive type. They don't extend from `BinaryOperator` or `BiFunction`.

For example, here's the definition of `IntBinaryOperator`:

```
@FunctionalInterface
public interface IntBinaryOperator {
     int applyAsInt(int left, int right);
}
```

That you can use instead of `BinaryOperator`.

Key Points

- Java 8 contains new functional interfaces to work with lambda expressions that cover the most common scenarios usages located in the `java.util.function` package.
  They are:
    - `Predicate`
    - `Consumer`
    - `Function`
    - `Supplier`
    - `UnaryOperator`
- These interfaces have versions that work with primitive values for `int`, `long` and `double`, and `boolean` (only for `Supplier`) just to avoid the cost of converting a wrapper class to its primitive value, for example, `Integer` to `int`.
- These interfaces take one argument (represented by the generic type `T`), but with the exception of `Supplier` (that doesn't take any arguments), they have versions that take two arguments called binary versions.
- A Predicate can be used anywhere you need to evaluate a `boolean` condition. Its function descriptor (method signature) is:
  ```
  T -> boolean
  ```
- It has primitive versions for `int`, `long` and `double`, for example, `IntPredicate`.
- A Consumer is an operation that accepts a single input argument and returns no result. Its functional descriptor is:
  ```
  T -> void
  ```
- It has primitive versions for `int`, `long` and `double`, for example, `IntConsumer`.
- A Function is an operation that takes an input argument of a certain type and produces a result of another type. Its functional descriptor is:
  ```
  T -> R
  ```
- It has a lot of primitive versions for `int`, `long` and `double`. We can divide them into three types.
- To indicate that the function returns a generic type and takes a primitive argument, the interface is named **XXXFunction**, for example, `IntFunction`.
- To indicate that the function returns a primitive type and takes a generic argument, the interface is named **ToXXXFunction**, for example, `ToIntFunction`.
- To indicate that the function takes a primitive argument and returns another primitive type, the interface is named **XXXToYYYFunction**, where XXX is the argument type and YYY is the return type, for example, `IntToDoubleFunction`.

- A `Supplier` is an operation that takes no arguments, but it returns some value. Its functional descriptor is:

```
() -> T
```

- It has primitive versions for `int`, `long`, `double` and `boolean`, for example, `IntSupplier`.
- `UnaryOperator` is a specialization of the `Function` interface (in fact, this interface extends from it) for when the argument and the result are of the same type. So its functional descriptor is:

```
T -> T
```

- It has primitive versions for `int`, `long` and `double`, for example, `IntUnaryOperator`.
- A `BiPredicate` represents a predicate that takes two arguments. Its functional descriptor is:

```
(T, U) -> T
```

- This interface doesn't have primitive versions.
- A `BiConsumer` represents a consumer that takes two arguments. Its functional descriptor is:

```
(T, U) -> void
```

- It has primitive versions for `int`, `long` and `double`. They take one object and a primitive value as a second argument. So the naming conventions change to `ObjXXXConsumer`, where XXX is the primitive type, for example, `ObjIntConsumer`.
- A `BiFunction` represents a function that takes two arguments of different type and produces a result of another type. Its functional descriptor is:

```
(T, U) -> R
```

- It has primitive versions that take generic types as arguments and return `int`, `long` and `double` primitive types, with the naming convention `ToXXXBiFunction`, where XXX is the primitive type.
- A `BinaryOperator` is a specialization of the `BiFunction` interface (in fact, this interface extends from it) for when the arguments and the result are of the same type. Its functional descriptor is:

```
(T, T) -> T
```

- It defines two static methods:

```
static <T> BinaryOperator<T> minBy(
          Comparator<? super T> comparator)
static <T> BinaryOperator<T> maxBy(
          Comparator<? super T> comparator)
```

- It has primitive versions for `int`, `long` and `double`, for example, `IntBinaryOperator`.

Self Test

1. Given:

```
public class Question_10_1 {
      public static void main(String[] args) {
           Predicate<String> p1 = t -> {
                System.out.print("p1");
                return t.startsWith(" ");
           };
           Predicate<String> p2 = t -> {
                System.out.print("p2");
                return t.length() > 5;
           };
           p1.and(p2).test("a question");
```

```
      }
}
```

What is the result?
A. `p1`
B. `p2`
C. `p1p2`
D. `false`
E. Compilation fails

2. Which of the following interfaces is a valid primitive version of `BiConsumer<T, U>` ?
A. `IntBiConsumer`
B. `ObjLongConsumer`
C. `ToLongBiConsumer`
D. `IntToDoubleConsumer`

3. Given:

```
public class Question_10_3 {
      public static void main(String[] args) {
            IntUnaryOperator u1 = i -> i / 6;
            IntUnaryOperator u2 = i -> i + 12;
            System.out.println(
                  u1.compose(u2).applyAsInt(12)
            );
      }
}
```

What is the result?
A. `24`
B. `14`
C. `4`
D. `2`
E. Compilation fails

4. Which of the following statements is true?
A. A `Consumer` takes a parameter of type `T` and returns a result of the same type.
B. `UnaryOperator` is a specialization of the `Operator` interface.
C. The `BiFunction` interface doesn't have primitive versions.
D. A `Supplier` represents an operation that takes no arguments, but it returns some value.

5. Given:

```
public class Question_10_3 {
      public static void main(String[] args) {
            Supplier<Boolean> s = () -> {
                  Random generator = new Random();
                  int n = generator.nextInt(1);
                  return n % 2 == 0;
            };
            System.out.println(s.getAsBoolean());
      }
}
```

What is the result?
A. `true`
B. `false`
C. Sometimes `true` , sometimes `false`
D. Compilation fails

6. Which of the following interfaces is a valid primitive version of `BiPredicate<T, U>` ?
A. `IntBiPredicate`
B. `ObjBooleanPredicate`
C. `ToLongBiPredicate`
D. `BiPredicate` doesn't have primitive versions

7. Which of the following primitive version of `Function` returns a generic type while taking a `long` argument?
A. `ToLongFunction`
B. `LongFunction`
C. `LongToObjectFunction`
D. There's no primitive version with this characteristic

8. Which of the following statements is true?
A. The `BinaryOperator` interface extends from the `BiFunction` interface.
B. The `BiSupplier` interface only takes one generic argument.
C. The `Supplier` interface doesn't define any default methods.
D. `minBy` and `maxBy` are two default methods of the `BinaryOperator` interface.

Open answers page

---

09. Lambda Expressions 11. Method References